from where to store his files, given a set of nodes which can execute his programs, to the major decision of whether to join the network at all. This model can be applied in the analysis of these problems.

Received January 1975; revised September 1976

References

1. Bulfin, R.L., and Unger, V.E. Computational experience with an algorithm for the lock box problem. Proc. ACM Annual Conf., Aug. 1973, Atlanta, Ga., pp. 16–19.
2. Casey, R.G. Allocation of copies of a file in an information network. Proc. AFIPS 1972 STCC, Vol. 40, AFIPS Press, Montvale, N.J., 1972.
3. Chu, W.W. Optimal file allocation in a multicomputer information system. Information Processing '68, North-Holland Pub. Co., Amsterdam, 1969, pp. 1219–1225; also *IEEE Trans. Comptrs. C-18*, 10 (Oct. 1969), 885–889.
4. Gonzales, A.R.H. Solution of the traveling salesman problems by dynamic programming on the hypercube. M.S. Th., School of Industrial Management, M.I.T., Cambridge, Mass., May 1962.
5. Ellwein, L.B. Fixed charge location allocation problems with capacity and configuration constraints. Ph.D. Diss., Dept. Indust. Eng., Stanford U., Stanford, Calif., August 1970.
6. Ellwein, L.B. A flexible enumeration scheme for zero-one programming. *Oper. Res. 22*, 2 (Feb. 1974), 145–150.
7. Kahn, R.E. Resource-sharing computer communication networks. *Proc. IEEE, 61* (Nov. 1972), 1397–1407 (special issue on computer communications).
8. Khumwala, B.M. An efficient branch and bound algorithm for the warehouse location problem. *Manage. Sci. 18*, 12 (Aug. 1972), B-718–B-731.
9. Kleinrock, L. Models for computer networks. Proc. Int. Conf. on Communication, Boulder, Colo., June 1969, pp. 2.9–2.16.
10. Levin, K.D. Organizing distributed data bases in computer networks. Ph.D. Diss. U. of Pennsylvania, Phila., Pa. 1974.
11. Levin, K.D., and Morgan, H.L. Optimizing distributed data bases—a framework for research. Proc. AFIPS 1975 NCC, Vol. 44, AFIPS Press, Montvale, N.J., pp. 473–478.
12. Levin, K.D. Two algorithms for optimal file assignments in heterogeneous computer networks. Tech. Rep. 75-08-02, Dept. Decision Sci., The Wharton School, U. of Pennsylvania, Phila., Pa.,
13. Manne, A.S. Plant location under economies of scale—decentralization and computations. *Manage. Sci. 11* (Nov. 1964), 213–225.
14. Mahmoud, S.A. Resource allocation and file access control in distributed information networks. Ph.D. Diss. Syst. Eng. Dept., Carleton College, Northfield, Minn., 1975.
15. Whitney, V.K.M. A study of optimal file assignment and communication network configuration. Ph.D. Th. U. of Michigan, Ann Arbor, Mich., 1970.

Artificial Intelligence/ Language Processing

C.A. Montgomery Editor

A Comparison of Tree-Balancing Algorithms

J.-L. Baer and B. Schwab
University of Washington, Seattle

Several algorithms—height-balance (i.e. AVL and extensions), weight-balance (i.e. BB and WB), and total restructuring—for building balanced binary search trees are compared. The criteria for comparison encompass theoretical aspects (e.g. path lengths) and implementation independent and machine/algorithm-dependent measures (e.g. run time). A detailed analysis of code is also presented at a level believed to be language- and compiler-independent. The quality of the resulting trees and the overhead spent on building them are analyzed, and some guidelines are given for an efficient use of the methods. If insertion and subsequent queries are the only operations of interest, then "pure" AVL trees present the overall best qualities.

Key Words and Phrases: binary search trees, AVL trees, weight-balanced trees, path length, analysis of algorithms, information storage and retrieval

CR Categories: 3.7, 3.72, 3.74, 5.31

This work was supported in part by NSF Grant GJ-41164 and by a Burroughs Fellowship Grant to B. Schwab.

Author's addresses: J.L. Baer, Department of Computer Science, University of Washington, Seattle, WA; B. Schwab, Boeing Aerospace Company, Seattle, WA 98124.

1. Introduction

Binary search trees represent an important technique for handling structures such as files and directories, dictionaries, and symbol tables. Nievergelt [8] gives a comprehensive survey of the methods used to create and modify binary search trees as well as some pertinent analytical results on the efficiency of these methods. Figure 1 provides a classification of these trees and references to previous works in this area. In this paper we restrict ourselves to the study of non-weighted binary search trees.

The random growth of a binary search tree can lead in the worst case to a (linked) linear list. Hence several algorithms have been devised to balance or restructure the tree while it is being built, i.e. to keep it close to its optimal form. The three main methods for doing this—height-balance, weight-balance, and total restructuring—are introduced in the next section. We stress some aspects of the algorithms not covered previously in the literature.

The main thrust of this paper is to compare these various methods. Path-length measures have already been used to assess the "goodness" of the resulting tree. We introduce some implementation-independent factors which affect the work generated to produce the trees. It will be seen in Section 3 that theoretical consideration as well as results of simulation experiments are sufficient to justify balancing techniques but not discriminating enough to make a judgment between them.

In the next section, we try to be more precise about the concept of work. Naturally we have to relinquish some of the independence from the implementation. However, by looking at some timing experiments as well as at the monitoring of primitive operations, we feel that we can draw some conclusions about the respective values of the algorithms.

In particular, we show that a height-balance technique, the AVL tree construction, is the most efficient when the operations on trees are limited to insertion and queries. Furthermore we demonstrate that the idea of compromising between obtaining random and balanced binary search trees by relaxing balancing criteria is not justified, since the loss in average path length is not compensated by the decrease in the number of balancing operations.

2. Balancing Methods

Definitions

A *binary tree* is a finite set of nodes either empty, in which case we call it a nil node or a nil tree, or the triple $T(T_L, r, T_R)$, where r is a special node called the *root*, and T_L and T_R are, respectively, the *left* and *right subtrees of r*. Each node s, except a nil node, is the *father* of its *left* and *right sons* (the roots of T_L^s and T_R^s). A node with two nil-node sons is a *leaf*.

In this paper (as well as in the computer representations of the algorithms to follow), each non-nil node has four fields: *LEFTLINK* and *RIGHTLINK*, which are pointers to its left and right sons, a *BALANCE* field depending on the algorithm, and a *DATA* field. In *binary search trees*, the *DATA* field is some alphanumeric information, or *key*, such that, for any non-nil node $s \in T$,

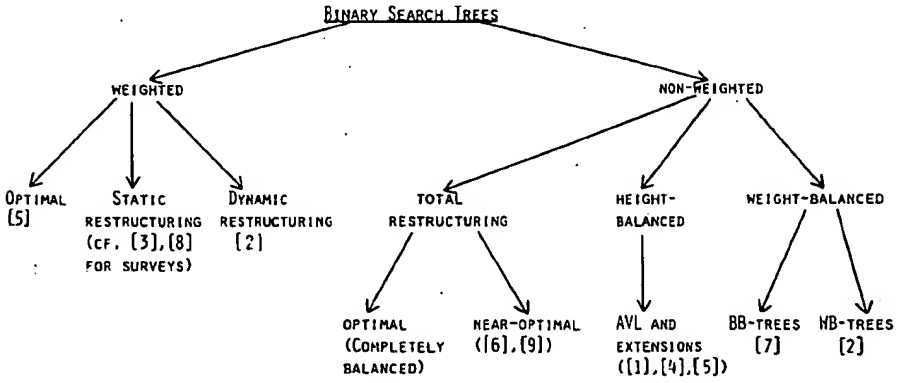
$$\forall s_i \in T_L^s, \quad key(s_i) < key(s),$$
$$\forall s_j \in T_R^s, \quad key(s_j) > key(s).$$

Notice that we forbid equal keys. A nil node is pointed to by a *NIL* pointer from its father, and its fields are meaningless.

The *level* of a node is 1 if the node is the root; otherwise it is the level of its father plus 1. The level of a tree is the maximum of the levels of its non-nil nodes (i.e. the maximum level of its leaves). The *height* of a node is 0 for a nil node and the maximum of the heights of its sons plus 1 otherwise. The height of the tree is the height of its root (as well as its maximum level). Note also that a leaf has height 1.

The *path length* of a tree is the sum of the levels of

Fig. 1. Techniques to handle binary search trees.



its nodes, or equivalently the sum over all nodes of the number of nodes in the path from the root to the node. The *average path length* is the path length divided by the number of nodes.

A binary search tree is *completely balanced* if all levels (except possibly the highest) are full. Such a tree is *optimal* with respect to minimal path length. A binary search tree is *balanced* if it is built according to the height and weight balance algorithms described below. A balancing algorithm will be *top-down* if it restructures the tree during the insertion of a new leaf and *bottom-up* (although maybe only partially) if it has to retrace the path of insertion.

Mechanics of Balancing

The construction of a random binary search tree, i.e. one without balancing, is trivial. The key of the insertion node, which is necessarily a leaf, is matched with the key of the root. If it is smaller the process is repeated recursively with the left subtree, and if it is larger with the right subtree until a nil node is encountered. This is the place of insertion for the new leaf (*LEFTLINK* and *RIGHTLINK* are set to *NIL, DATA* to key) for which the immediately previous non-nil node encountered is the father. The obvious link connection from the father is performed, and this implies keeping track of the father of the inserted node and the direction of the path of insertion (labeled *F* and *FGO* in figures and algorithms).

Balancing is performed through transformations called *single* or *double rotations*. They are activated by the balancing algorithms under criteria which vary with each method. However, the resulting tree structures are the same and are sketched in Figure 2 (symmetric rotations naturally do exist) with their respective link changes (three for a single, five for a double rotation).

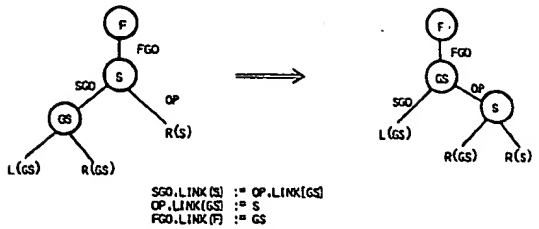
Height-Balance Algorithms

AVL trees [1,5] (named after their inventors, the two Russian mathematicians Adel'son-Vel'skiy and Landis) and their extensions [4] are built according to a height-balance algorithm. A binary search tree is *AVL* if the heights of the left son and of the right son of any node in the tree do not differ by more than 1. The BALANCE field can indicate this with two bits (+1: higher right son; 0: equal heights; −1: higher left son). The bottom-up algorithm for constructing an AVL tree consists of three phases:

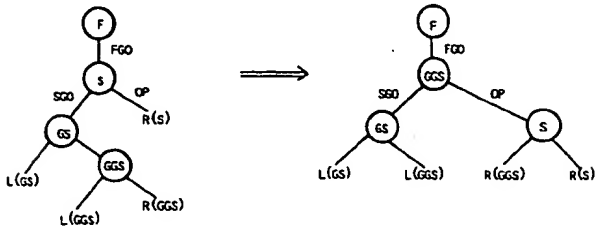
1. Find the place of insertion and the critical node.
2. Modify the *BALANCE* fields between the critical node (excluded) and the new leaf.
3. Balance at the critical node if necessary; otherwise modify its balance field.

The critical node found in phase 1 is the last node on the path of insertion of *BALANCE* nonzero, or the root if no such node exists. Hence phase 1 is similar to the construction of a random binary search tree with this added critical node checking. Phase 2 implies a

Fig. 2. Rotations.



(A) Single rotation (F (father), S (son), GS (grand son), SGO direction of insertion, OP opposite of SGO)



(B) Double rotation (GGS (great grand son))

bottom-up algorithm. The *BALANCE* modification is immediate and depends on the direction of the path of insertion. In phase 3, balancing occurs if the subtree whose root is the critical node becomes more imbalanced, i.e. if the direction of the path of insertion and the present *BALANCE* coincide. In this case, calling the critical node *S* with reference to Figure 2, if *S* and its son *GS* have keys both larger (or smaller) than the new leaf, i.e. if $SGO = GSGO$, then we have a single rotation; otherwise a double rotation occurs. If the subtree becomes less imbalanced, a modification of the *BALANCE* field of the critical node is sufficient. Finally, if the root was the critical node with balance 0, its balance field has to be modified.

Foster has extended AVL trees [4] by allowing a difference in height of δ ($\delta \geq 1$). The algorithm is as above, but now, for reasons of efficiency, the search for the critical node is best handled by phase 2. The rationale is that in the AVL case all of the information necessary is in the visited node, but now the heights of both sons have to be checked. Since one of them is not on the insertion path, one wants to minimize the accessing of nodes which are not of interest. Foster's idea was to trade-off increased path length versus a diminishing number of rotations. As we see later, this is not necessarily a good idea.

Weight-Balance Algorithms

In these algorithms the criteria for balancing are based on the cardinalities of the subtrees (denoted $|T^s|$ for a tree of root s).

A bounded-balance tree of balance α, *BB*(α) *tree* [7], has, for each node s,

$\alpha \le \rho_s = (|T_L^s| + 1)/(|T^s| + 1) \le 1 - \alpha,\ 0 \le \alpha \le \frac{1}{2}$.

A top-down algorithm for $0 \le \alpha \le 1 - \frac{1}{2}\sqrt{2}$ is described in the above reference and follows the general line of the WB tree construction given below. The criterion for balancing involves the computation of the ratio ρ at each node on the path, and a rotation is performed (dictated by the weight of the subtree and the direction of path of insertion) if ρ is out of limits. Special care is needed for $\alpha > \frac{1}{4}$ and $|T^s| = 2$, i.e. a "small" tree, and the rotations applied in this case will be called *leaf rotations* (two or three link changes). This "small" tree concept can be generalized for any α, saving some checking [9].

WB trees [2], or weight-balanced trees, have been introduced mainly to be applied to weighted trees. However, the algorithm is still correct when all weights are 1. WB trees are defined dynamically through either a top-down or a bottom-up algorithm. Here we consider only the former. As in BB(α) trees, the *BALANCE* field of a node s is $|T^s|$. The criterion for rotation is to minimize the path length at the father of S (F in Figure 2). In general terms, a single rotation would be called (cf. Figure 2(a)) if

$$weight\ (GS) + |T^{GGS}| > weight\ (S) + |T^S_{OP}|$$

(where OP is the opposite direction of SGO) and a double rotation (cf. Figure 2(b)) if

$$2\ weight\ (GGS) + |T^{GGS}_L| + |T^{GGS}_R| > weight\ (S) + |T^S_{OP}|\ .$$

However, with equal weights of 1, both criteria become

$$|T^{GGS}| > |T^S_{OP}|$$

since $1 + |T^{GGS}_L| + |T^{GGS}_R| = |T^{GGS}|$.

As for AVL trees, single and double rotations are chosen according to the keys of S and its son on the path of insertion. Similarly a difference of γ ($\gamma \ge 0$) can be introduced as a parameter in the balancing criterion for the same expected trade-off as before.

The top-down construction of a WB tree proceeds essentially like that of a random binary search tree. In addition, the following actions are taken during the visit of each node on the insertion path (except for the first two nodes which are handled differently since there can be no rotation until three nodes have been visited):

—A test for balance is performed.
—If the balance criterion is met, a (single or double) rotation is applied. This involves links and *BALANCE* field modifications.
—The *BALANCE* field of the visited node is modified (i.e. incremented by 1 if there was no rotation) and bookkeeping operations are performed to keep track of the nodes and associated directions needed for the next test for balance.

A detailed Pascal implementation of this algorithm and of all those presented in this paper can be found in [9].

Total Restructuring

While the term balancing is not justified for this method, the total restructuring of a binary search tree into an optimal one at some intervals to be determined is certainly worth investigating [6]. Several criteria to call for a restructuring come immediately to mind, and at least two of them are quite appealing [9].

(1) Keep track of the actual and optimal path lengths, $A(T)$ and $O(T)$. After each insertion check if $A(T) \ge (1 + \theta)\ O(T)$ $(0 \le \theta < 1)$, and if so restructure.

(2) Assume a restructuring has just taken place. Let $k = \lceil \log_2 |T| \rceil$[1] be the height of the tree. When $\phi 2^{k-1}$ $(0 \le \phi < 1)$ paths of insertion longer than $(\beta + k)$ (β is an integer greater than or equal to 1) are encountered, a restructuring is performed. The rationale here is that the kth level is generally not full and that 2^{k-1} nodes are needed to fill it. If the tree is degrading rapidly, restructuring will take place to fill that level.

In the series of experiments described below, we found that with $\beta = 1$ and various values of ϕ, criterion (2) was in general superior to criterion (1). In the following we refer to the algorithm with criterion (2), $\beta = 1$, a starting value of $k = 4$ for the testing in order to prevent restructuring of very small trees, and a value of $\phi = 0.05$ as algorithm COM. Later on, ϕ will be varied.

As pointed out by Martin and Ness [6], the restructuring can be done in place in $O(n)$ time (where $n = |T|$).

Experimental Data

As mentioned earlier, we performed a series of experiments to try to rank the methods. Two types of simulation runs were performed:

(1) *Experiment R*, which involved trees built from 10 sets of a 1000 randomly chosen keys. (We also took intermediary measures at 500 and 750 nodes.)

(2) *Experiment W*, which involved trees built from the alternating set of keys (1, 1000, 2, ...). The letter W is to draw attention to the fact that it is a worst case for the random and COM constructions and a "bad" case for the work involved in building the other trees. The alternating set was chosen instead of the ordered set because it produces single and double rotations instead of only single rotations in the balancing algorithms.

3. Implementation-Independent Measurements and Comparisons

While comparing the methods to construct binary search trees, we have to take into account two factors: the "goodness" of the resulting tree, i.e. its closeness to the optimal binary search tree, and how much work was

[1] $\lceil x \rceil$ is the smallest integer greater than or equal to x. $\lfloor x \rfloor$ is the largest integer smaller than or equal to x. In this paper all logarithms are base 2.

involved in order to obtain the binary search tree. The first measure is related to subsequent searching in the tree, while the second is an approximation of the overhead incurred in building the tree.

The common metric for a searching mechanism is the number of comparisons used to find a match (or no match) for a given key. Translated into a tree structure, it implies path length measures. The worst-case measure is the *maximum level* that a tree can achieve. For the algorithms of Section 2 and a tree of n nodes, we have

optimal tree	: $\lceil \log n \rceil$	$\Rightarrow O(\log n)$
random tree	: n	$\Rightarrow O(n)$
AVL	: $1.44 \log(n+2) - 0.328$ (Fibonacci tree)	$\Rightarrow O(\log n)$
$BB(1-\frac{1}{2}\sqrt{2})$	: $2(\log(n+1)-1)$	$\Rightarrow O(\log n)$
WB-tree	: $2.32(\log(n+1)-1)$ (BB($\frac{1}{4}$))	$\Rightarrow O(\log n)$
COM (ϕ)	: $\lceil \log n \rceil + 1 + \phi 2^{\lceil \log n \rceil - 1}$	$\Rightarrow O(n)$

All these results but the last one can be found in the references already cited. The worst tree constructed by the COM method is obtained when, after a restructuring, $(1 + \phi 2^{k-1})$ nodes are inserted on the same path. However, one should remark that these long "tails" disappear periodically when the tree grows. Table I (column 1) shows the maximum levels obtained in experiments R and W. The maximum level for COM (0.05) is, according to the above formula,

$$\lceil \log 1000 \rceil + 1 + 0.05 \times 2^{\lceil \log 1000 \rceil - 1} \doteq 37.$$

Interestingly enough this is the same figure as for $n = 750$, and in the experiment W (a real worst case for COM) the maximum at $n = 750$ was 32, larger than the 30 obtained at $n = 1000$. Somewhere in between the maximum level had reached 37 (in fact several times).

Also of interest is the *average path length*. Since a random search tree has an average path length of 1.39 log $(n + 1)$ [5], all binary search trees constructed by the above methods will have average path lengths of $O(\log n)$. No analytical results more precise than this order of magnitude have yet been obtained. By looking at Table I (column 2), we can see that all methods yield similar results for experiment R. Notice that although COM (ϕ) produces long tails in experiment W, there is no degradation in the average path length.

The value of the balancing algorithms is evident from these first two results. The (theoretically expected) savings of close to 40 percent in average path length are present. More important, worst cases of linear path length are either avoided or intermittent (in the case of COM).

Table I. Implementation-Independent Measures. $n = 1000$, $\lceil \log n \rceil = 10$, optimal (average) path length 8.98.

	1 Max. Level	2 Av. path length	3 Av. insert. path	4 Rotations S	D	Leaf	Total
Experiment R							
Random	22.6	12.14	12.14	–	–	–	0
AVL ($\delta = 1$)	12.	9.20	9.73	230	231	–	461
BB ($\alpha = 1 - \frac{1}{2}\sqrt{2}$)	12.6	9.26	9.70	84	64	278	426
WB ($\gamma = 0$)	12.	9.16	9.70	267	248	–	515
COM (ϕ = 0.05)	11.8	9.06	9.68	–	–	–	12.8
Experiment W							
Random	1000	500.5	500.5	–	–	–	–
AVL ($\delta = 1$)	12	9.27	11.51	371	617	–	988
BB ($\alpha = 1 - \frac{1}{2}\sqrt{2}$)	13	9.36	12.32	225	400	416	1041
WB ($\gamma = 0$)	16	9.68	10.67	356	633	–	989
COM (ϕ = 0.05)	30	9.20	18.11	–	–	–	74

Table II. Path Length versus Rotation Trade-Off.

		1 Max. Level	2 Av. path length	3 Av. insert. path	4 Rotations S	D	Leaf	Total
Experiment R								
AVL	$\delta = 1$	12	9.20	9.73	230	231	–	461
	$\delta = 2$	13.2	9.37	9.88	104	103	–	207
	$\delta = 5$	16.7	10.16	10.60	23	25	–	48
BB	$\alpha = 1 - \frac{1}{2}\sqrt{2}$	12.6	9.26	9.70	84	64	278	426
	$\alpha = .25$	14.2	9.46	10.01	173	69	–	242
	$\alpha = .15$	16.8	10.03	10.51	96	21	–	117
WB	$\gamma = 0$	12	9.16	9.70	267	248	–	515
	$\gamma = 1$	12.8	9.25	9.79	131	136	–	267
	$\gamma = 4$	14.2	9.49	10.04	50	60	–	110
COM	$\phi = 0.05$	11.8	9.06	9.68				12.8
	$\phi = 0.1$	12.9	9.10	9.75				8.7
	$\phi = 0.25$	12.6	9.14	9.86				4.6
Experiment W								
AVL	$\delta = 1$	12	9.27	11.51	371	617	–	988
	$\delta = 2$	13	9.28	12.49	370	617	–	987
	$\delta = 5$	16	9.28	15.45	370	614	–	984
BB	$\alpha = 1 - \frac{1}{2}\sqrt{2}$	13	9.36	12.32	225	400	416	1041
	$\alpha = .25$	17	9.42	15.22	332	703	–	1035
	$\alpha = .15$	26	10.11	22.54	485	489	–	974
WB	$\gamma = 0$	16	9.68	10.67	356	633	–	989
	$\gamma = 1$	16	9.69	11.66	356	631	–	987
	$\gamma = 4$	16	9.69	14.62	356	630	–	985
COM	$\phi = 0.05$	30	9.20	18.11				74
	$\phi = 0.1$	26	9.12	26.56				45
	$\phi = 0.25$	124	15.54	52.29				20

Work Measures

We now turn our attention to measures of the work needed to build the tree, but only from an implementation-independent viewpoint. The first factor of interest is the (average) *insertion path*, i.e. the amount of searching done before inserting a new node. Since later balancings might modify this path, and in all cases shorten it, the insertion path is longer than the path length (it is the same for random trees). Table I (column 3) shows again that, with the exception of COM, all methods behave similarly. In experiment W we see that COM has an average insertion path almost double that of the others; this can be explained easily by the formation of the long "tails" mentioned earlier.

The insertion path was introduced because it is one of the adequate measures of the work involved for example in the detection of the next node to visit and the check for balance as described in Section 2.

We can associate also with the check for balance procedure a measure of the *locality* of the algorithm,

i.e. the number of nodes visited during an insertion. For the random and AVL constructions this number is similar to the insertion path (although for the AVL extension this is not true anymore, as seen in Section 4). For the weight balance algorithms, nodes which are not on the insertion path are needed. In the BB case, the ρ computation calls always for the left son; hence, assuming that the insertion path is as likely to proceed in either direction, one has to access a node not on the path on every other check. In the WB tree the *BALANCE* of the son of S which is not on the insertion path is needed at every check. Therefore it would appear that if AVL's locality is normalized to 1, BB's locality will be approximately 1.5 and WB's will be 2. But in the weight balance algorithms we have the relation

$$BALANCE \text{ (left son)} + BALANCE \text{ (right son)} + 1 = BALANCE \text{ (node)}.$$

Hence if we were trying to restrict the number of visited nodes, as e.g. in a virtual memory or cache environment, the nodes on the insertion path would be sufficient.

The locality of the COM algorithm is very much linked to the number of restructurings m. In fact it can be computed as L = insertion path + $\sum_{i=1}^{m} |T_i|$, where T_i is the tree at the ith restructuring. In experiment R, L was about $13n$ (i.e. 1.4 times the insertion path and consequently approximately 1.4 times the locality of the AVL construction), and in experiment W, L was $40n$. This latter figure can be obtained analytically since, as we show below, we can obtain the number of restructurings in that case.

The *number of rotations* certainly appears to be an influential factor. Very few analytical results can be derived on the number of rotations per insertion. Note that there is at most one rotation per insertion in the AVL algorithm, while possibly more than one will occur in the weight-balance constructions. Table I (column 4) summarizes the experimental results. The BB algorithm seems slightly superior in experiment R, and experiment W certainly justifies its name for this measure.

For the COM algorithm, it is possible to derive the number of restructurings needed in the worst case. Assume that we have just performed a restructuring. The earliest time the next one is going to occur is after the insertion of $\lceil \phi\ 2^{k-1} \rceil + 2$ nodes. (Recall the worst case for the maximum level.) So in order to fill level k of the tree we need $2^{k-1}/(\lceil \phi\ 2^{k-1} \rceil + 2)$ restructurings. For example, starting the algorithm (experiment W) with $k_0 = 4$, $2^{k_0-1} = 8$, there is no restructuring until six nodes have been entered. In fact when the first 15 nodes are entered, we will have $m_0 = 3$ restructuring. Therefore the total number of restructurings m is

$$m = m_1 + \sum_{k=5}^{\lfloor \log n \rfloor} \frac{u}{\lceil \phi\ u \rceil + 2} + \frac{n - v}{\lceil \phi\ v \rceil + 2}$$

where $u = 2^{k-1}$ and $v = 2^{\lfloor \log n \rfloor - 1}$, and in our case

$$m = 3 + \left(\left\lfloor \frac{16}{3} \right\rfloor + \left\lfloor \frac{32}{4} \right\rfloor + \cdots + \left\lfloor \frac{256}{15} \right\rfloor + \left\lfloor \frac{1000 - 512}{28} \right\rfloor \right)$$
$$= 3 + (5 + 8 + \cdots + 17 + 17) = 74.$$

Again we can see that COM is quite sensitive to the worst case.

Trading Off Increased Path Length for Decreased Work

As was said in Section 2 we can relax the balancing criterions. Table II shows the same figures as Table I with different values of the parameters. For small changes of the parameter (e.g. δ and γ by 1, α and ϕ by 0.05), we see an increase in path length of less than 3 percent while the number of rotations is approximately reduced by half (in experiment R). In experiment W, however, path length and number of rotations remain the same, but the insertion path is increased by 10 percent or more. Hence for that latter case allowing more imbalance is detrimental. But for the former, it appears that less work is required to build a tree of almost equal quality. Jumping at this conclusion is erroneous, as will be demonstrated in the next section.

4. Comparison of Methods

From the preceding section it is apparent that the balancing methods yield trees of equal quality. If one looks only at measures like the insertion path and the number of rotations, again there is not much difference. In order to be more discriminating, we are obliged to delve more deeply into the details of the algorithms. Our goal is to answer the questions:

—What is the "best" balancing algorithm?

—Assuming that most trees are random, what is the ratio of the number of insertions to the number of subsequent queries which makes a balancing worthwhile?

—What is the value of varying the parameters of balancing criteria?

The analysis that we present now should by no means be considered as an absolute judgment on the "complexity" of the algorithms. It is simply an attempt at ranking the algorithms based on some experimental evidence and a (nonrigorous) analysis of code. We feel that the code is of the same efficiency for all methods. The analysis is as much as possible machine and language independent, but some implementation considerations may enter the discussion.

We start by analyzing the random algorithm. Let I be the average insertion path. Our unit of "time" will be a "*move*" along the path which corresponds to a key comparison, two pointer settings (father and son), and a direction setting. The number M_R of moves of that type in the construction of a random binary search tree of n nodes is $M_R = n\ (I - 1)$, with I in this case being

also the average path length.

The only other "work" performed by the algorithm is the insertion of new nodes, which implies getting a new node from a list of available space (or incrementing an index), setting the data field and pointer links, and linking the new leaf to its father. This is of the same order of magnitude as a move. Hence the cost of the algorithm is $C_R = n\,I$.

Although some other work will be done in conjunction with this part of the algorithms, the code corresponding to the search of the place of insertion in a random tree appears in all algorithms. In all tree-searching algorithms, the number of moves to find the place of insertion is $M = n\,(I - 1)$. This value for M is slightly too small because it does not take into account the overhead due to the modularity of the program (procedure calls). As before we also add the insertion time, i.e. $M' = n\,(I - 1) + n = n\,I$. We consider now the added work needed in each algorithm.

AVL Analysis

Phase 1 of the AVL construction does the searching and insertion (cost M'_A) and the detection of the critical node. To obtain the latter, we need one comparison per node on the insertion path, except for the root and the leaf, and if the balance is not zero one more comparison and four pointer settings. The latter should occur approximately 35 percent of the time (cf. [4, 5] for analysis and justification). Since we have one comparison per node and a little less than a move $\frac{1}{2}$ of the time, we can approximate the cost of phase 1 by:

$$P_1 = M'_A + 0.5n\,(I - 2),$$
$$P_1 = n\,(1.5\,I - 1).$$

In phase 2, the balance fields between the critical node (excluded) and the leaf (excluded) must be changed. The same analysis as before [4, 5] shows that on the average there are a little less than 2 nodes (1.8 to be exact) between the critical node and the leaf. The balance field change itself has a cost slightly higher than that of a move. Therefore $P_2 = 1.8 \times 1.25n = 2.25n$.

Finally, in phase 3 either a balance change occurs at the critical node or we perform a rotation. Quoting again [4, 5] (cf. also Table I), we see that we have a balance change a little over half of the time. A single rotation (three link settings and two balance fields settings) is approximately one move, while a double rotation (five link settings and one comparison plus two balance settings) corresponds to two moves. Hence

$$P_3 = 0.5 \times 1.25n + 0.25n\,(1 + 2) = 1.37n,$$

and

$$C_A = P_1 + P_2 + P_3,$$
$$C_A = n\,(1.5\,I + 2.62). \qquad (1)$$

To justify this analysis, we consider the results of experiment R. We saw (Table I) that the AVL average insertion path was 9.7, yielding $C_A = 17.17n$. Setting a normalized C_R to 1 (instead of $12.14n$), we have $C_A = 17.05/12.14 = 1.41$.

Table III. CPU Times.

		Experiment R	Experiment W
RAN		0.58	>10
AVL		0.96	1.06
Extended AVL	$\delta = 1$	1.49	1.83
	$\delta = 2$	1.55	2.13
	$\delta = 5$	1.64	2.75
WB	$\gamma = 0$	1.71	1.92
	$\gamma = 1$	1.72	2.11
	$\gamma = 4$	1.74	2.64
BB	$\alpha = 1 - \frac{1}{2}\sqrt{2}$	2.20	2.84
	$\alpha = 0.25$	2.36	3.68
	$\alpha = 0.15$	2.32	5.34
COM	$\phi = 0.05$	0.93	3.61
	$\phi = 0.10$	0.77	2.58
	$\phi = 0.25$	0.71	2.90

In order to test our assumptions, we timed the algorithms (cf. Table III) on a CDC 6400, discounting the overhead of reading in the keys. For experiment R the average of T_A/T_R was $T_A/T_R = .96/.58 = 1.65$. This is certainly in accordance with our analysis. The fact that our prediction is quite accurate should not be overemphasized. The order of magnitude is the important fact.

WB Analysis

The search and insertion times are as in the AVL construction, i.e. $M'_W = nI$. The test for balance costs approximately 1.5 moves (four settings, two comparisons) and has to be done on the whole insertion path except for the first two nodes. Hence $B_W = 1.5\,n\,(I - 2)$.

The bookkeeping implied in either the rotation procedures or the check for balance corresponds to an increment in the balance field and five pointer settings, again a little more than a unit move. It has to be done on all nodes but the first two (handled differently) and the leaf. Therefore $G_W = 1.25\,(I - 3)$. The rotations are slightly more expensive than in the AVL tree because of the changes in the balance fields. A single rotation is about two moves and a double rotation about four. Hence the rotation work is $R_W = 0.25\,(2 + 4)n = 1.5 \cdot n$, and finally

$$C_W = n(3.75\,I - 5.25). \qquad (2)$$

Testing as above, $C_W = 31.12/12.14 = 2.56$, and the timing experiment yielded $T_W/T_R = 1.75/.58 = 3.02$.

A possible reason for our conservative estimate is that in the check for balance we called a function for finding an opposite direction, while in the analysis we counted it as an online segment of code.

BB-Analysis

The BB algorithm for $\alpha = 1 - \frac{1}{2}\sqrt{2}$ follows the same line as the WB one. However, there is one major difference: the check for balance includes a division, two comparisons, and two additions in opposition to a comparison and a subtraction in the WB case. The

amount of time spent in a division being very much machine dependent, we can only assert that the BB algorithm will take longer than the WB. Our timing experiments yielded $T_B/T_R = 2.20/.58 \simeq 3.79$. Since the CDC 6400 performs division rapidly, timing on slower machines might show degradation of this ratio.

COM-Analysis

In the COM algorithm, a move is the same as in the random construction. Hence $M_c' = nI$. The check for balance consists of one incrementation per node on the path and two comparisons per insertion, i.e. overall a fraction of the cost M_c'. Most of the overhead of the COM algorithm is then spent in the restructuring. Since the traversal of the trees to be restructured is very much implementation-dependent, there is no hope of analyzing its cost as with the other methods. Just as an idea of what can be obtained, our timing experiments yielded $T_c/T_R = .93/.58 \simeq 1.62$. But because in the worst case T_c is multiplied by a factor of 4, compared to 1.1 or 1.2 for the balancing algorithms, this method is not a competitive one.

A First Conclusion

From the above analysis it appears that AVL is the best algorithm. If one assumes that worst-case trees, or trees close to them, can arise, then all balancing algorithms (but maybe COM) are of interest. On the other hand, if it is certain that only random trees may occur, one should know at which point the overhead due to balancing is overcome by the quality of the resulting tree. Considering first the AVL tree, if q queries are performed after the original insertion, then the (average) cost will be

$$Q_R = (q + 1)C_R, \qquad \text{for the random tree}$$

and

$$Q_A = C_A + qI = C_A + 0.8qC_R. \qquad \text{for the AVL tree}$$

if the ratio of average path lengths is independent of the size of the tree. As soon as q is such that $(0.2q + 1)C_R > C_A$, the AVL tree construction becomes beneficial. Our analysis tends to show that q must be at least 3.

A similar reasoning yields a value for q of 8 for the WB tree and certainly over 10 for the BB tree. Note again that these figures are independent of the size of the tree (formulas (1) and (2)). Before answering the question of the significance of the weight-balance algorithms, since the height balance ones behaves better, we look at the effect of the variations of the balancing parameters.

Extended AVL Tree

In the extended AVL, the contents of the balance field is the height of the node. The algorithm for extended AVL is slightly different from the "pure" one since the critical node search is now done in a bottom-up phase. The top-down phase 1 stacks the path of insertion. This extra bookkeeping of pointers and directions (three assignments and one increment) costs a little less than a move. Hence $P_1' = 1.75n(I - 1) + n$. In phase 2 we go back up the tree, checking for the critical node and changing the balance fields. The critical node detection implies the difference of heights between the two sons of a given node (notice now a degradation in the locality of the algorithm which is not correctable as in the weight-balance methods). Taking the case $\delta = 1$ (i.e. comparing with the "pure" AVL), we have 2.8 checks for a critical node which involves initialization, moving up the stack (.5 move), finding the heights of both sons and updating the height of one node (1 move), and checking for a critical node and a possible rotation (1 move). If the node is not critical, we go back up the stack (1.75 move).

$$P_2' = (2.8(0.5 + 1 + 1) + 1.8 \times 1.75)n = 10.15n.$$

Finally, the rotations are slightly more time consuming because of the height changes; hence $P_3' = 0.25(2 + 3)n = 1.25n$, and the cost of the height-balance algorithm is

$$C_H = n(1.75I + 10.65). \tag{3}$$

For experiment R this becomes $C_H \simeq 27.62/12.14 = 2.27$, and the timing experiments yielded $T_H/T_R = 1.50/.58 = 2.58$, again close enough for our purposes.

Comparing formula (3) and formula (1) shows that in all cases the pure AVL method is better, since in formula (3) the insertion path has a larger coefficient.

Returning now to formula (3), we see that the rotation cost P_3' is only $1.25n$. Increasing δ decreases this number, but will also increase the path length and the length of the bottom-up pass. For example, with $\delta = 2$ we found out that[2] (with the obvious notation):

$$\begin{aligned} P_1'(2) &= P_1'(1) + 1.75 \times 0.015nI \simeq P_1'(1) + 0.26n, \\ P_2'(2) &= P_2'(1) + (3 - 2.8) \times (2.5 + 1.75)n \\ &\simeq P_2'(1) + 0.85n, \\ P_3'(2) &= P_3'(1)/2.25 \simeq P_3'(1) - 0.70n. \end{aligned}$$

Hence the increase due to the imbalance is about $.41n$ in unnormalized fashion or 0.04 in normalized fashion. The increase in time was very small, as expected (0.06 instead of the expected 0.02). And repeating the same analysis with $\delta = 5$ yielded .12 expected time increase for .15 real increase.

Independently of the accuracy of the prediction, we can see that the savings in rotation time is always a small percentage of the total running time. Therefore keeping $\delta = 1$ is preferable.

Weight-Balance Extensions

We can perform the same type of analysis for the WB and BB algorithms. According to the derivation of

[2] We are indebted to C.C. Foster for pointing out that the critical node, when $\delta \neq 1$, is the node furthest from the root at which either a rotation is needed or where the height of the son on the path of insertion is not greater than the height of the other son.

formula (2), a decrease in rotation would save very little. For example, letting $\gamma = 1$ saves $0.75n$ in R_W (cf. Table III) and adds $3.75(I' - I)$, where I' is the new insertion path (cf. formula (2)), i.e. in our experiment $3.75 \times 0.09n = .34n$. Hence a total savings of about $.4n$, or in CPU time an expected 0.02. In fact measurements showed an increase of 0.01 in time, again a very accurate prediction. For the BB construction, the increase due to a longer path insertion will be larger because of more divisions, but on the other hand short cuts can be taken in the number of checks when the subtree on the path of insertion becomes "small" [7]. Our measurements showed that for $\alpha = .25$ and $\alpha = .15$ there was very little difference in CPU time for experiment R, but a substantial time increase was apparent in experiment W.

The CPU times for a variation of ϕ showed a slight decrease for COM times as ϕ became larger. This is in accordance with the fact that the work on the path of insertion counts for less than in the other methods and that restructurings are costly. Making ϕ large, however, would be extremely dangerous if a tree close to the worst case were to happen (cf. Table II, $\phi = 0.25$).

We are now ready to summarize the results from Sections 3 and 4 and make some judgment about the respective qualities of the methods.

5. Summary and Conclusion

Quality of Trees for Searching

This is a path length measure. As shown in Section 3, all four methods yield an average path length very close to the optimal one. Because of possible long paths, the total restructuring method is less reliable.

Work Involved in Building a Tree

The AVL construction presents less overhead. For all trees the elegant method presented in Section 2 should be followed. If the number of subsequent queries is larger than three per node, then AVL trees should be recommended over random trees. Similarily WB trees can be used if there are at least 8 queries per insertion, and for BB trees this ratio is at least 10 with a machine performing fast divisions.

If, besides insertions and queries, one wants to perform splitting and concatenation of AVL trees, then the balance field must contain the height of a node [5]. If furthermore one wants to use the tree as a ranked list, a field similar to the one used in BB and WB trees is necessary. Therefore weight-balance algorithms might be more competitive for some applications and should not be discarded entirely. This is even more true of WB algorithms, which can also be used efficiently for weighted trees [2].

Relaxing the Balancing Criterion

Relaxing the balancing criterion gives a slightly worse tree for subsequent searchings and, most important, does not save any work. For the AVL tree, there is a non-negligible loss in time and locality. For WB, extended AVL, and BB trees, the cost of building trees is stationary for random trees but increases substantially for degenerate trees.

Therefore we can present two important conclusions:

—Balancing methods are worthwhile even if the nodes in resulting trees are going to be queried only a few times each (on the average).

—The best balancing methods are the "purest," i.e. those involving the strictest balancing criteria. This is true of the quality of the resulting tree and the cost of building it.

Finally, we might suggest that some comparisons of that type could be done for other data structures such as the implementation of priority queues by heaps or leftist trees.

Received September 1975, revised March 1976

References

1. Adel'son-Velskiy, G.M., and Landis, Y.M. An algorithm for the organization of information. *Doklady Akad. Nauk, SSSR* (Moscow) *16*, 2 (1962), 263-266. Translation, OTS, JPRS, 17, 137, Dept. of Commerce, Washington, D.C.
2. Baer, J.-L. Weight-balanced trees. Proc. AFIPS 1975 NCC, Vol. 44, AFIPS Press, Montvale, N.J., pp. 467-472.
3. Domesle, R. On the construction of weighted binary search trees. Masters Th., U. of Washington, Seattle, Wash., June 1975.
4. Foster, C.C. A generalization of AVL trees. *Comm. ACM 16*, 8 (Aug. 1973), 512-517.
5. Knuth, D. *The Art of Computer Programming, Vol. 3: Sorting and Searching*. Addison-Wesley, Reading, Mass., 1973.
6. Martin, W.A., and Ness, D.N. Optimizing binary trees grown with a sorting algorithm. *Comm. ACM 15*, 2 (Feb. 1972), 88-93.
7. Nievergelt, J., and Reingold, E.M. Binary search trees of bounded balance. *SIAM J. Computing 2*, 1 (March 1973), 33-43.
8. Nievergelt, J. Binary search trees and file organization. *Computing Surveys 6*, 3 (Sept. 1974), 195-207.
9. Schwab, B. Comparison of balancing algorithms for binary search trees. Masters Th., U. of Washington, Seattle, Wash., June 1975.